Concurrency Patterns

(Mostly) for Library Authors

26 May 2015

Aaron Schlesinger
Sr. Engineer, Iron.io

About me

- Database & backend systems engineer at Iron.io
- Writing Go for ~2 yrs, JVM for a while during/before that
- Distributed systems at Zynga, StackMob, PayPal, now Iron
- C -> C++ -> Python/PHP -> -> Java -> Scala -> Go. I'm finally happy

"It's almost too easy to write concurrency bugs in Go"

Because Go has powerful concurrency primitives.

I'm discussing today how to use them responsibly.

I said library authors, but this talk is aimed toward anyone who writes reusable code.

Today

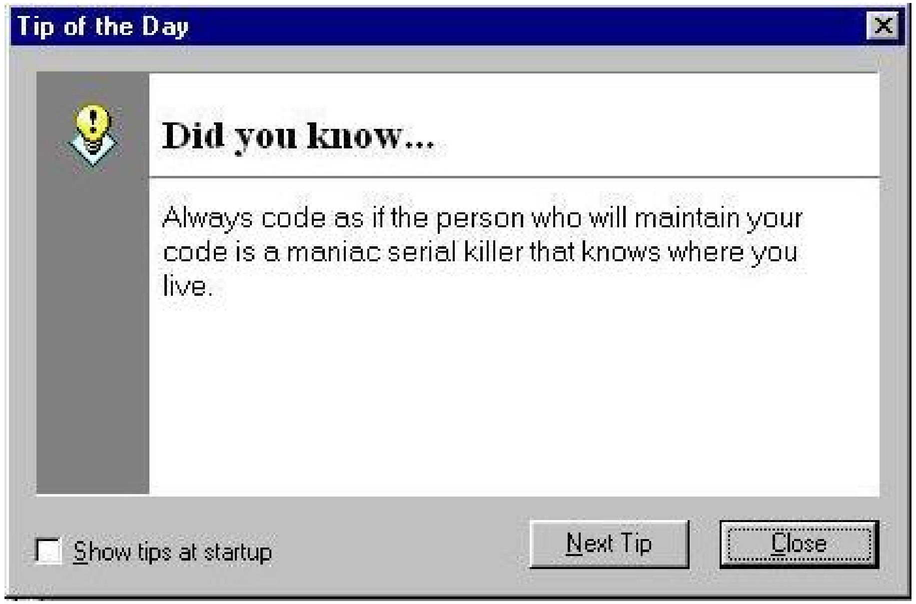
Conventions & patterns for:

- `mutex`, `chan`, `select`, and `sync.WaitGroup`
- Timeouts (http://blog.golang.org/go-concurrency-patterns-timing-out-and), cancellation and net.Context (http://godoc.org/golang.org/x/net/context)
- For-select loops

Something old

Locks have a time and a place. We found one at Iron.io (building a distributed DB).

If you use locks:



But prefer sharing memory by communicating.

Go-specific conventions

- `defer` unlocking wherever possible
- Document ownership in your **public** interface
- If mutual exclusion crosses a public interface boundary, abstract it

Also, good conventions already exist from other communities (e.g. lock order conventions).

Documentation

```
package main

import "errors"

var ErrNotReserved = errors.New("not reserved")

// RequestState is an in-memory table to track the state of all requests.
// States can be any positive number.
type RequestState interface {
    // GetAndReserve waits for the state to be available, then returns it.
    // After this call returns, you have ownership of this state. Use
    // the returned string as an ownership token to indicate your ownership of id
    // in later calls. Returns an error if communication with underlying storage fails.
    // If this func returns error, you do not have ownership.
    GetAndReserve(id string) (int, string, error)

    // SetReserved sets the state of the given request ID, given the ownership ID.
    // doesn't set the state and returned ErrNotReserved if the state wasn't reserved
    // by the given ownership ID. After this call returns successfully, you no
    // longer have ownership of id and ownershipID is invalidated.
    SetReserved(id, ownershipID string, state int) error
}
```

Channels

```
Share memory by communicating.
```

This means passing messages.

- Channels + goroutines are "easy" but powerful enough to build real systems
- They're first class for a reason. Use them by default
- When in doubt, ask why you *shouldn't* use them to communicate between goroutines

Specifically

- Document channel communication across `func` boundaries
- Enlist the compiler. Use directional channels
- Don't return a chan unless the `func` is a generator (https://talks.golang.org/2012/concurrency.slide#25)
- `close` is a useful signal to callers. Use it and document it

Example

```
package main

import "sync"

// WatchChanges will watch the state of the given request. ch will send after
// each request state change and will be closed after the request is removed from
// the request state database.
//
// WatchChanges sends on ch from the same goroutine as the caller and will deadlock
// if there is no goroutine already listening on it.
//
// Returns ErrNotFound if the request is not reserved at call time.
// If any error is returned, WatchChanges will neither do any operations on ch.
func WatchChanges(reqID string, ch chan<- int) error

// WatchAll watches for all events on the given request.
//
// The WaitGroup will be done after the request is reserved, and the channel
// will send on each state change, then be closed when the request is released.
//
// The channel will send from a new, internal goroutine, which you are not responsible
// for managing.
func WatchAll(reqID string) (*sync.WaitGroup, <-chan int)
```

Interlude 1

- Documentation may establish contracts or invariants that code can't or won't
- That being said, the remainder of this talk shows has mostly runnable code
- Imagine that it has proper documentation in it

Select

Channels of channels:

- We have (probably more than) a few posts on how/why to use them
- Particularly, listening on channels-of-channels in a `select` enables a goroutine to offer many "services"
- This pattern can be helpful for introspecting the state of a goroutine

Introspection

Trivial example:

```
func doWork(calc <-chan chan<- int, cur <-chan chan<- int, history <-chan chan<- []int, stats ch
an<- int) {
    var all []int
    i := 0
    for {
        select {
        case ch := <-calc:
            i := rand.Int()
            ch <- i
            all = append(all, i)
        case ch := <-cur:
            ch <- i
        case ch := <-history:
            ch <- all
        }
        i++
        stats <- i
    }
}
```

Run

WaitGroup

In general, if you're waiting for a chan to close or receive a `struct{}`, think about using a WaitGroup.

Passing these around as signals of behavior can also help you write deterministic tests.

Notification of an event

Trivial example.

```
package main

import "sync"

// startLoop starts a loop in a goroutine. the returned WaitGroup is done
// after the first loop iteration has started
func startLoop(n int) *sync.WaitGroup {
	var wg sync.WaitGroup
	go func() {
		first := true
		for {
			if first {
				wg.Done()
				first = false
			}
			// do some work here
		}
	}()
	return &wg
}
```

Revisiting fan-in

(A better image is at https://talks.golang.org/2012/concurrency.slide#28 (https://talks.golang.org/2012/concurrency.slide#28))

Why?

If you can do work concurrently, do it. There's no excuse not to.

How I'm defining fan-in:

- A pattern to gather results from 2 or more goroutines
- A procedure you can follow to take sequential code and convert it to concurrent

Details

- Read about it under "Fan-out, fan-in" section at https://blog.golang.org/pipelines (https://blog.golang.org/pipelines).
- `sync.WaitGroup` and a few channels make fan-in simple & understandable enough to convert non-concurrent code to concurrent code with this pattern.
- In many cases, you can get an easy latency win without changing an API.

Sequential datastore queries

```
func GetAll() []int {
    ints := make([]int, 10)
    for i := 0; i < 10; i++ {
        ints[i] = datastoreGet()
    }
    return ints
}
```

Run

Concurrent datastore queries

```
func GetAll() []int {
    wg, ch := getWGAndChan(10) // get a waitgroup that has 10 added to it, and a chan int
    for i := 0; i < 10; i++ {
        c := make(chan int)
        go datastoreGet(c)  // sends an int on c then closes after sleeping <= 1 sec
        go func() {
            defer wg.Done() // mark this iteration done after receiving on c
            ch <- <-c       // enhancement: range of c if >1 results
        }()
    }
    go func() {
        wg.Wait() // wait for all datastoreGets to finish
        close(ch) // then close the main channel
    }()
    ints := make([]int, 10)
    i := 0
    for res := range ch { // stream all results from each datastoreGet into the slice
        ints[i] = res // GetAll can be a generator if you're willing to change API.
        i++           // that lets you push results back to the caller.
    }
    return ints
}
```

Run

Interlude 2

In production, don't grind to a halt when a channel doesn't send.

In Go, we can use `time.Timer` to time out channel receives.

But we're gonna go a step further.

Timeouts

But, `net.Context` (https://blog.golang.org/context) has a nice interface in front of timers.

- I'm cheating here. I said everything would be done with the standard library
- We're *mostly* using contexts as a timer, but it provides a nice `interface` with a few extra bells & whistles.
- That's my excuse

Using contexts

- If you are waiting for a channel, don't wait forever
- Ideally, all goroutines take a context
- Contexts add more control and testability

`net.Context` is a good universal tool for timeouts/cancellation in a large codebase.

Polling a queue

```
func main() {
    ctx, cancel := context.WithTimeout(context.Background(), 500*time.Millisecond)
    defer cancel()
    ch := make(chan string)
    go poll(ctx, ch) // sends dequeued results on ch. stops & closes ch when ctx.Done() receives
    for polled := range ch {
        fmt.Println(polled)
    }
}
```

Run

Contexts in a distributed system

The Tail at Scale.

- Jeff Dean talk/paper. I originally saw it at a Ricon 2013 Talk (https://www.youtube.com/watch?v=C_PxVdQmfpk)
- Hedged requests: do a few identical GET (e.g. no side effects) requests, cancel remaining requests after first returns

Rob showed a variant in https://talks.golang.org/2012/concurrency.slide#50 (https://talks.golang.org/2012/concurrency.slide#50)

Adding cancellation

With `net.Context`!

```
func main() {
    ch := make(chan int)
    ctx, cancel := context.WithTimeout(context.Background(), 10*time.Millisecond)
    defer cancel()
    for i := 0; i < 10; i++ {
        // get (not shown) sleeps for a random duration <= 100ms,
        // then sends a random int on ch. stops if ctx.Done() receives.
        go get(ctx, ch)
    }
    select {
    case i := <-ch:
        fmt.Printf("got result %d\n", i)
    case <-ctx.Done():
        fmt.Println("got no result")
    }
}
```

Run

That was the naïve implementation

But, it's not too hard to get "fancy"

- Don't send 2nd request until 1st is past watermark (95th percentile expected latency)
- Cancel in-flight requests (pass context to RPC subsystem)
- Target-target communication (pass info on other in-flight requests over RPC)

For-select loops

Running a (possibly infinite) loop inside a goroutine, selecting on 1 or more channels at each iteration.

Lots of possible applications:

- Event loops
- GC
- Synchronizing on state (like an actor)

Patterns

- ack before and after real work is done. testing is easier and rate limiting/backpressure is easy
- if you're ticking, build acks into `time.Ticker`
- `net.Context` for cancellation
- `sync.WaitGroup` for started and stopped loops

A for-select poller

```
func poll(ctx context.Context) (*sync.WaitGroup, *sync.WaitGroup, <-chan string) {
    var start, end sync.WaitGroup
    start.Add(1)
    end.Add(1)
    ch := make(chan string)
    go func() {
        defer close(ch)
        defer end.Done()
        start.Done()
        for {
            time.Sleep(5 * time.Millisecond)
            select {
            case <-ctx.Done():
                return
            case ch <- "element " + strconv.Itoa(rand.Int()):
            }
        }
    }()
    return &start, &end, ch
}
```

Driving the poller

```
func main() {
    ctx, cancel := context.WithTimeout(context.Background(), 10*time.Millisecond)
    defer cancel()
    mainCh, wg := makeThings(10) // make a chan string and a wg that has 10 added to it
    for i := 0; i < 10; i++ {
        start, _, ch := poll(ctx)
        start.Wait()
        go func() {
            defer wg.Done()
            for str := range ch {
                mainCh <- str
            }
        }()
    }
    go func() {
        wg.Wait()
        close(mainCh)
    }()
    printCh(mainCh) // loops on mainCh until it's closed
}
```

Run

Notes

- The poller is missing the ack
- We have a small utility at Iron to add acks to `time.Ticker` and `time.Timer`
- Exercise left to the reader

Conclusion

Go has really good concurrency abstractions built into the language.

I believe we (the community) are starting to build good patterns & tools to *responsibly* build on them.

If you take one thing away

Use `net.Context` somewhere in your codebase.

Or, at least try it.

It's simple and powerful, and follows the "Go way."

If you take two things away

Add reading and understanding Go Concurrency Patterns (https://talks.golang.org/2012/concurrency.slide) .

Thank you

Aaron Schlesinger
Sr. Engineer, Iron.io
aaron@iron.io (mailto:aaron@iron.io)
http://github.com/arschles (http://github.com/arschles)
@arschles (http://twitter.com/arschles)